# ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКИХ УСКОРИТЕЛЕЙ ДЛЯ РЕШЕНИЯ ЗАДАЧ КЛАССА «DATA INTENSIVE»

**А.С. Колганов**

*ИПМ имени М.В. Келдыша РАН*

**Введение**

В последнее время все большую роль играют графические ускорители (GPU) в неграфических вычислениях. Потребность их использования обусловлена их относительно высокой производительностью и более низкой стоимостью.

Как известно, на GPU хорошо решаются задачи на структурных сетках, где параллелизм так или иначе легко выделяется. Но есть задачи, которые требуют больших мощностей и используют неструктурные сетки. В качестве примера таких задач можно назвать Single Shortest Source Path problem (SSSP) – задача поиска кратчайших путей от заданной вершины до всех остальных во взвешенном графе и Breadth First Search (BFS) – поиска в ширину в неориентированном графе. Последняя задача используется в качестве основного теста для рейтинга Graph500, который был создан для оценки производительности вычислительных систем при обработке большого количества данных.

Для решения данных задач с помощью центрального процессора (CPU) существует, по крайней мере, два известных алгоритма: алгоритм Дейкстры [1] и алгоритм Форда—Беллмана [2]. Также существуют параллельные реализации алгоритмов поиска в ширину и поиска кратчайшего пути в графе на GPU [3-6].

**Архитектура графического процессора**

За последнее время архитектура графических ускорителей сильно изменилась. Несомненно, это связано с их целевой направленностью, а именно использованием в неграфических вычислениях. Для таких целей был разработан модельный ряд GPU Tesla. Рассмотрим строение GPU на примере самой последней архитектуры Kepler на начало 2014 года [7-9].

Новый чип Kepler GK110 был разработан в первую очередь для пополнения модельного ряда Tesla. Его цель – стать самым быстрым параллельным микропроцессором в мире. Чип GK110 не только превышает по производительности чип предыдущего поколения Fermi, но и потребляет значительно меньше энергии.

GK110 в максимальной конфигурации состоит из 15 потоковых мультипроцессоров, называемых SMX, и шести 64 битных контроллеров памяти (см. Рис. 1). Основные особенности GK110 следующие:

- совершенно новая архитектура потокового мультипроцессора SMX;
- новая подсистема памяти, улучшены пропускные способности кэшей всех уровней, дополнительные возможности кэширования, а также значительно улучшена работа с глобальной памятью;
- аппаратная поддержка вычислительной способности (compute capability) 3.5 и ниже;
- поддержка динамического параллелизма – функции, которая позволяет потокам GPU динамически генерировать новые потоки, чтобы динамически адаптироваться к данным. Новая технология существенно упрощает параллельное программирование за счет применения GPU-ускорения к широкому спектру распространенных алгоритмов, таких как адаптивное уточнение сеток, быстрые мультипольные и многосеточные методы;
- Hyper-Q — функция, которая позволяет нескольким процессам CPU одновременно использовать ядра CUDA на одном GPU Kepler. Нагрузка на GPU значительно вырастает, уменьшается простой CPU и улучшается программируемость;
- GPU Direct – данная технология позволяет графическим процессорам обмениваться данными внутри единой сети, не вовлекая в этот процесс центральный процессор компьютера.

Каждый потоковый мультипроцессор SMX содержит 192 cuda-ядра для операций одинарной точности, 64 cuda-ядра для операций двойной точности, 32 специальных функциональных блока и 32 блока для загрузки и сохранения данных, четыре планировщика. Для всех SMX на GPU предусмотрен один общий кэш второго уровня, размер которого составляет 1.5 МБ. Также у каждого SMX есть свой собственный кэш первого уровня размером 64 КБ.

Для того чтобы использовать большие мощности GPU, необходимо отобразить вычислительно независимые участки кода на группы независимых виртуальных нитей. Каждая группа будет исполнять одну и ту же команду над разными входными данными. Для управления данной группой виртуальных нитей необходимо объединить их в блоки фиксированного размера. Данные блоки в архитектуре CUDA называются cuda-блоками.

Такой блок имеет три измерения. Все блоки объединяются в решетку блоков, которая также может иметь три измерения. В конечном счете, каждый cuda-блок будет обработан каким-либо потоковым мультипроцессором и каждой виртуальной нити будет сопоставлена физическая. Минимальной единицей

параллелизма на GPU является warp – группа из 32х независимых нитей, которые исполняются физически параллельно и синхронно.


Рис. 1. Структура чипа GK110

**Описание структуры данных**

Кратко рассмотрим структуру хранения неориентированного взвешенного графа. Граф задается в сжатом CSR (Compressed Sparse Row) формате. Данный формат широко распространен для хранения разреженных матриц и графов.

Для хранения графа с N вершинами и M ребрами необходимо три массива: xadj, adjncy, weights. Массив xadj размера N + 1, остальные два – 2*M, так как в неориентированном графе для любой пары вершин необходимо хранить прямую и обратную дуги.

Принцип хранения графа следующий. Весь список соседей вершины I находится в массиве adjncy с индекса xadj[I] до xadj[I+1] - 1. По аналогичным индексам хранятся веса каждого ребра из вершины I. Для иллюстрации на Рис.2. слева показан граф из 5 вершин, записанный с помощью матрицы смежности, а справа – в CSR формате.

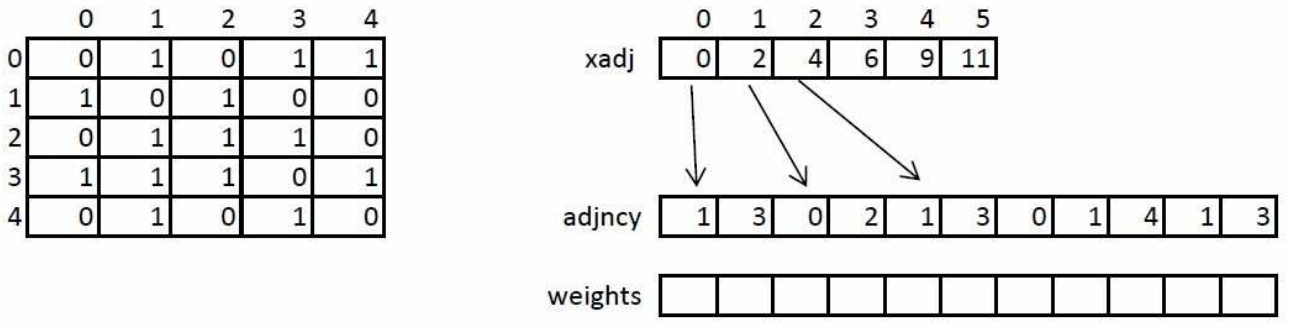


Рис. 2. Структура хранения графов

**Реализация алгоритма SSSP на GPU**

***Подготовка входных данных***

Для того чтобы увеличить вычислительную нагрузку на один потоковый мультипроцессор, необходимо преобразовать входные данные. Все преобразования можно разделить на два этапа:

- Расширение формата CSR

- Сортировка расширенного формата CSR

На первом этапе необходимо расширить формат CSR следующим образом: введем еще один массив startV, в котором будем хранить начала дуг. Тогда в массиве adjncy будут храниться их концы. Таким образом, вместо хранения соседей, будем хранить дуги. Пример данного преобразования на графе, описанном выше, на Рис.3.

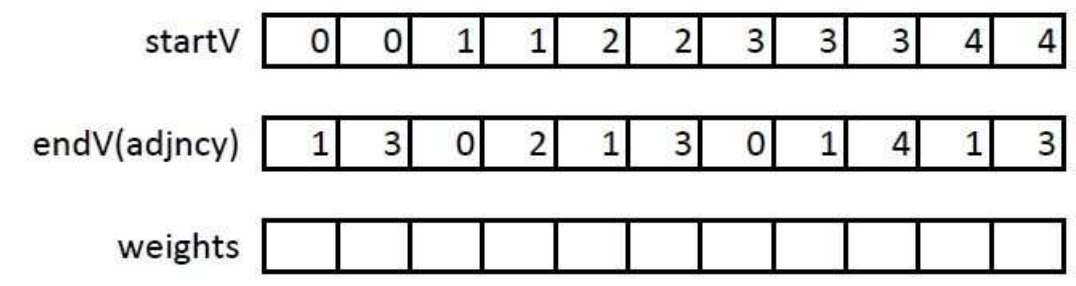


Рис.3. Преобразованный CSR формат

На втором этапе необходимо отсортировать полученные ребра так, чтобы каждая пара (U,V) встречалась ровно один раз. Таким образом, при обработке ребра (U,V) можно сразу обработать ребро (V,U), не считывая повторно данные об этом ребре из глобальной памяти GPU.

**Основное вычислительное ядро**

За основу для реализации на GPU взят алгоритм Форда—Беллмана [2]. Данный алгоритм хорошо подходит для реализации на GPU в силу того, что ребра можно просматривать независимо друг от друга и данные ребер и их весов расположены подряд, что улучшает пропускную способность глобальной памяти GPU. Рассмотрим основное вычислительное ядро (Листинг 1).

```
int k = blockIdx.x * blockDim.x + threadIdx.x;
if(k < maxV)
{
  double w = weights[k];

     unsigned en = endV[k];
  unsigned st = startV[k];

     if(dist[st] > dist[en] + w) // (*)
  {
     dist[st] = dist[en] + w;
     modif[0] = iter;
     }
      else if(dist[en] > dist[st] + w) // (**)
     {
           dist[en] = dist[st] + w;
           modif[0] = iter;
     }
}
```

**Листинг 1.** Вычислительное ядро для алгоритма SSSP

В данном ядре каждая нить обрабатывает два ребра (прямое и обратное), пытаясь улучшить расстояние по одному из них. Ясно, что оба условия if-блока не могут выполниться одновременно, так как они являются взаимоисключающими. В отличие от алгоритма Форда—Беллмана, где каждое ребро просматривается последовательно, в реализованном на GPU алгоритме может возникнуть ситуация «гонки» потоков – когда два или более потока заходят обновить одну и ту же ячейку dist[I]. Покажем, что в данном случае алгоритм останется корректным.

Предположим, что есть две нити K1 и K2, которые хотят обновить ячейку dist[I]. Это означает, что выполнилось условие (*) или (**). Возможны два случая. Первый – одна из двух нитей записала наименьшее значение. Тогда на следующей итерации для этих двух нитей условие будет ложно, а значение в ячейке dist[I] окажется минимальным. Второй – одна из двух нитей записала не минимальное значение. Тогда на следующей итерации для одной из нитей условие будет истинно, а для другой – ложно. Тем самым, результат будет одинаковый в обоих случаях, но достигнут за разное количество итераций.

Согласно оптимизированному варианту алгоритма Форда—Беллмана, если на какой-либо итерации не было изменений в массиве dist, то ответ задачи найден и выполнять следующую итерацию уже не нужно. На GPU для этих целей была введена переменная modif, в которую нити записывают номер текущей итерации.

Одна итерация – один запуск ядра. Реализация базового алгоритма на GPU состоит в следующем: в цикле запускается вычислительное ядро, далее считывается переменная modif и, если она не изменилась с

предыдущей итерации, то в массиве dist ответ задачи — кратчайший путь из заданной вершины до всех остальных.

**Оптимизация реализованного алгоритма SSSP**

Рассмотрим некоторые оптимизации, которые позволяют существенно улучшить производительность алгоритма SSSP, используя архитектуру Kepler.

***Использование текстурного кэша***

В данной задаче требуется постоянно обновлять и считывать массив расстояний dist. Данный массив занимает достаточно мало места в глобальной памяти GPU по сравнению с информацией о дугах и их весах. Например, для графа с количеством вершин $2^{20}$ (примерно 1 млн) массив dist будет занимать 8МБ. Несмотря на это, доступ к этому массиву осуществляется случайным образом, что плохо для GPU, так как порождаются дополнительные загрузки из глобальной памяти на каждый вычислительный warp. Для того чтобы минимизировать количество загрузок на один warp, можно использовать текстурный кэш.

Так как текстурный кэш доступен только на чтение, то для того чтобы им воспользоваться необходимо ввести две разные ссылки на один и тот же массив расстояний dist. Рассмотрим соответствующий код (Листинг 2).

```
__global__ void relax_ker (const double * __restrict dist, double *dist1,
… …)
{
     int k = blockIdx.x * blockDim.x + threadIdx.x + minV;
     if(k < maxV)
     {
          double w = weights[k];
          unsigned en = endV[k];
          unsigned st = startV[k];

          if(dist[st] > dist[en] + w)
          {
              dist1[st] = dist[en] + w;
              modif[0] = iter;
          }
          else if(dist[en] > dist[st] + w)
          {
              dist1[en] = dist[st] + w;
              modif[0] = iter;
          }
     }
}
```

**Листинг 2.** Оптимизация алгоритма SSSP – использование текстурного кэша

В результате внутри ядра все операции чтения ведутся с одним массивом, а все операции записи с другим. Но обе ссылки dist и dist1 указывают на один и тот же участок памяти GPU.

***Локализация данных для лучшего использования кэша***

Для лучшей работы описанной выше оптимизации необходимо, чтобы как можно дольше загруженные данные находились в L2 кэше. Доступ к массиву dist осуществляется по заранее известным индексам, хранящимся в массивах endV и startV. Для локализации обращений поделим массив dist на сегменты определенной длины, например, P элементов. Так как всего в графе N вершин, то получим (N / P + 1) разных сегментов. Далее, будем сортировывать ребра в данные сегменты следующим образом. В первую группу отнесем ребра, концы которых попадают в нулевой сегмент, а начала – в нулевой, затем в первый, во второй и т.д. Во вторую группу отнесем ребра, концы которых попадают в первый сегмент, а начала – в нулевой, затем в первый, во второй и т.д.

После данной перестановки ребер значения элементов массива dist, соответствующие, например, первой группе будут находиться в кэше максимально долго (длина кэш линии при этом будет равна P элементам), так как нити в первой группе будут запрашивать данные из нулевого сегмента для концевых вершин и нулевого, первого, второго и т.д. для начальных вершин.

**Реализация алгоритма BFS на GPU**

Также как и в задаче SSSP будем использовать те же самые преобразования для того, чтобы увеличить нагрузку на один SMX и снизить количество одинаковых данных, находящихся в глобальной памяти GPU. Для алгоритма BFS не нужны веса в графе. Стоит отметить, что необходимо хранить не кратчайшие расстояния, а номер уровня, в котором находится данная вершина. На графах большой связности количество уровней не превосходит размера самого минимального типа данных языка Си – unsigned char, что позволит сократить размер массива уровней в 8 раз по сравнению с массивом расстояний в алгоритме SSSP.

***Алгоритм на GPU***

За основу реализации был взят все тот же алгоритм Форда—Беллмана и ядро в рассмотренной задаче SSSP. Основное вычислительное ядро для BFS будет выглядеть следующим образом (Листинг 3):

```
if(k < maxV)
{
    unsigned en = endV[k];
    unsigned st = startV[k];
    if(levels[st] == iter)
    {
        if(levels[en] > iter)
        {
            levels_NR[en] = iter + 1;
            modif[0] = iter;
        }
    }
    else if(levels[en] == iter)
    {
        if(levels[st] > iter)
        {
            levels_NR[st] = iter + 1;
            modif[0] = iter;
        }
    }
}
```

**Листинг 3.** Вычислительное ядро для алгоритма BFS

Идея алгоритма в следующем. Пусть массив levels изначально заполнен максимальным для выбранного типа значением (255 для unsigned char). Будем передавать в ядро номер текущей итерации — iter. Далее необходимо просмотреть все ребра и проверить, является ли начальная или конечная вершина текущим родителем, то есть принадлежит ли она уровню iter. Если да, то необходимо пометить противоположную вершину просматриваемой дуги значением на единицу больше, чтобы «включить» эту вершину в список родителей на следующей итерации. Также как и в SSSP, вводится переменная modif, показывающая необходимость продолжения разметки графа.

Приведенный код (Листинг 3) уже содержит примененные оптимизации в задаче SSSP — использование текстурной памяти для массива levels и использование другой ссылки levels_NR, указывающей на тот же самый участок памяти, необходимой для записи; перестановка для лучшей локализации данных в кэше.

***Использование динамического параллелизма для оптимизации алгоритма BFS***

Для сильно связных графов в алгоритме BFS на начальных и конечных итерациях количество вершин, которые принадлежат определенному уровню не велико. Поэтому можно вместо просмотра всех ребер, просматривать массив вершин, и если какая-либо вершина принадлежит текущему уровню, то выполнять разметку ее потомков.

Для того, чтобы реализовать описанный выше алгоритм, нужно использовать динамический параллелизм CUDA. Алгоритм состоит в следующем. Каждой нити будет сопоставлена одна вершина из массива всех вершин. Если данная вершина принадлежит текущему уровню разметки, то данная нить запускает вычислительное ядро (Листинг 3), которое выполняет разметку только для данной вершины. Тем самым, можно существенно сократить количество считываемых данных из глобальной памяти GPU и количество просматриваемых вершин и ребер.

**Анализ полученных результатов**

Для тестирования использовались синтетические неориентированные RMAT-графы [10], которые хорошо моделируют реальные графы из социальных сетей и Интернета. Рассматриваемые графы имеют среднюю связность 32, количество вершин является степень двойки. В Таблице 1 приведены графы, на которых проводилось тестирование.

Таблица 1. Характеристики тестируемых графов

| Количество вершин 2^N | Количество вершин | Количество дуг | Размер массива dist для SSSP, МБ | Размер массивов ребер и весов для SSPS, МБ | Размер массива levels для BFS, МБ | Размер массива ребер для BFS, МБ |
|---|---|---|---|---|---|---|
| 14 | 16 384 | 524 288 | 0,13 | 3 | <0,125 | 2 |
| 15 | 32 768 | 1 048 576 | 0,25 | 6 | <0,125 | 4 |
| 16 | 65 538 | 2 097 152 | 0,5 | 12 | <0,125 | 8 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 17 | 131 072 | 4 194 304 | 1 | 32 | 0,13 | 16 |
| 18 | 262 144 | 8 388 608 | 2 | 64 | 0,25 | 32 |
| 19 | 524 288 | 16 777 216 | 4 | 128 | 0,5 | 64 |
| 20 | 1 048 576 | 33 554 432 | 8 | 256 | 1 | 128 |
| 21 | 2 097 152 | 67 108 864 | 16 | 512 | 2 | 256 |
| 22 | 4 194 304 | 134 217 728 | 32 | 1024 | 4 | 512 |
| 23 | 8 388 608 | 268 435 456 | 64 | 2048 | 8 | 1024 |
| 24 | 16 777 216 | 536 870 912 | 128 | 4096 | 16 | 2048 |

Из-за использования самого маленького типа данных для хранения значений уровня в алгоритме BFS для графа с количеством вершин $2^{20}$ необходимо 1МБ для хранения массива levels, в то время как для такого же графа в задаче SSSP необходимо 8МБ.

Тестирование проводилось на сервере с установленным в нем GPU NVIDIA GTX Titan [12], у которого 14 SMX и 2688 CUDA-ядер и CPU Intel core i7 3го поколения с частотой 3,4ГГц и 8МБ кэша. Вместо времени показателем производительности будем считать количество дуг, обработанных за секунду (TEPS). В данном случае необходимо поделить полученное время на количество дуг в графе. В качестве конечного результата бралось среднее значение по 32 точкам. Для компиляции использовались компиляторы Intel 13й версии и NVCC CUDA 5.5 с флагами –O3 –arch=sm_35. Результаты тестирования алгоритма SSSP приведены на Рис. 4.

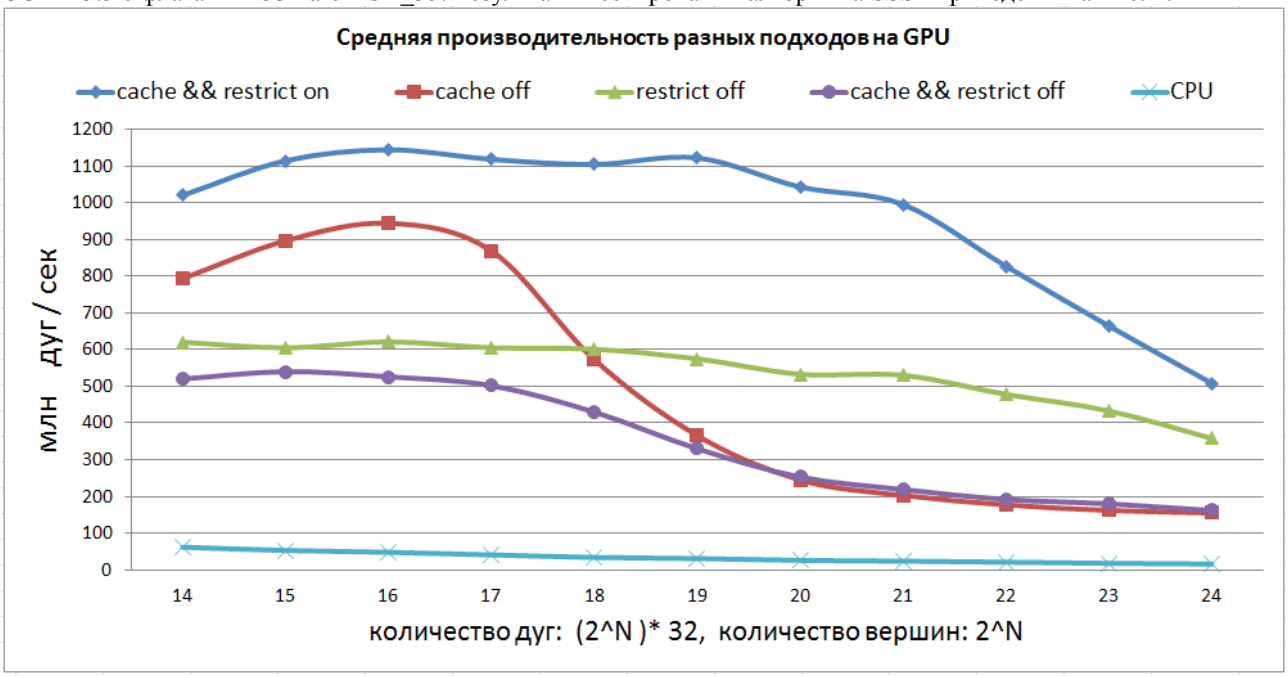


Рис.4. Средняя производительность разных подходов реализации алгоритма SSSP

На графике (см. Рис.4) изображены кривые средней производительности для следующих алгоритмов:

- cache && restrict on — алгоритм GPU со всеми оптимизациями (1);
- cache off — алгоритм GPU без оптимизации перестановок для улучшенного кэширования (2);
- restrict off — алгоритм GPU без оптимизации текстурного кэша (3);
- cache && restrict off — базовый алгоритм GPU без оптимизаций (4);
- CPU — базовый алгоритм на центральном процессоре.

Из приведенного графика (см. Рис.4.) видно, что каждая из оптимизаций по отдельности влияет на производительность на разных графах. Использование текстурного кэша (2) позволяет существенно улучшить скорость обработки маленьких графов. Это связано прежде всего с размером L2 кэша GPU. Использование перестановки данных (3) дает прирост на больших графах, так как улучшается кэширование. Применение двух подходов одновременно позволяет ускорить базовый алгоритм в 2 раза на маленьких графах и в 2-3 раза на больших. Максимальная производительность достигается на графе с $2^{19}$ вершин и равна 1,1 GTEPS.

Результаты тестирования алгоритма BFS приведены на Рис. 5.

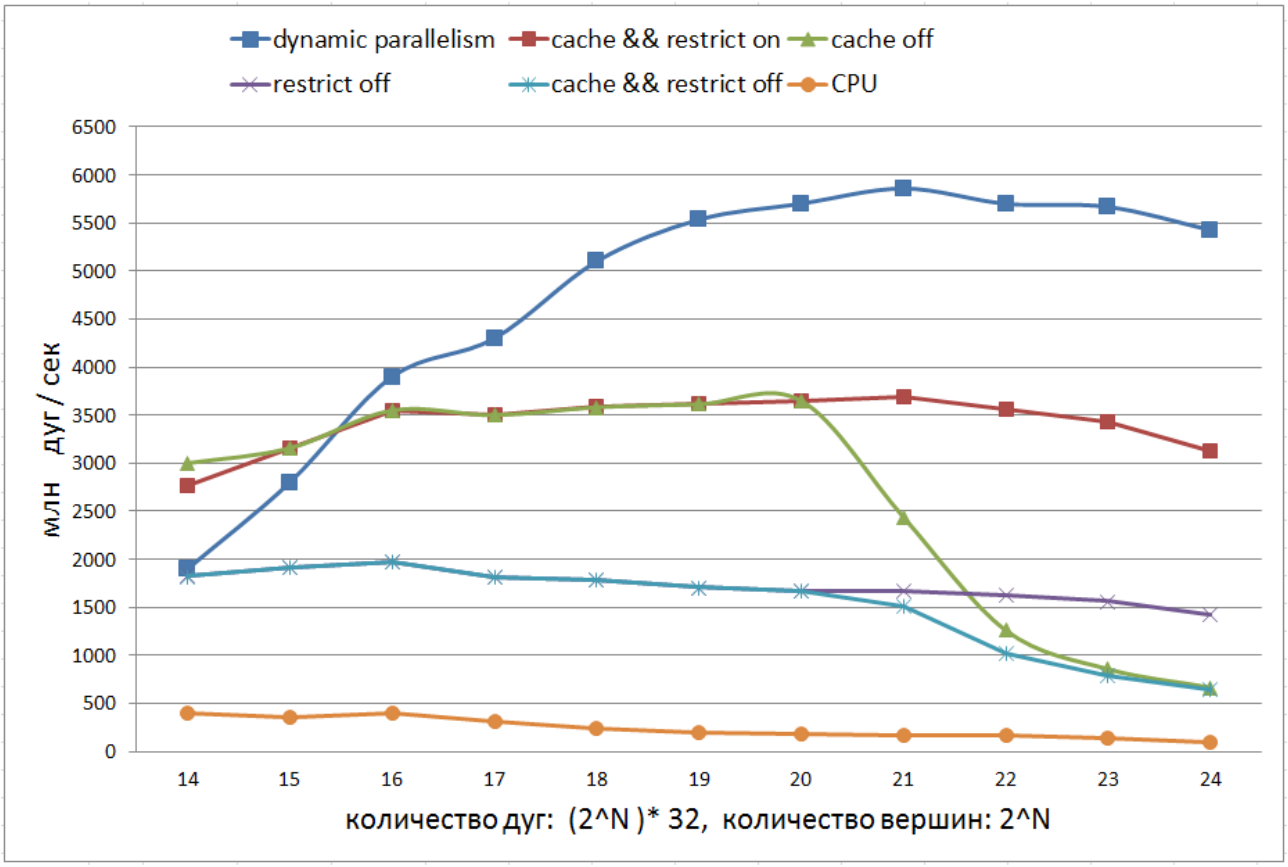


Рис.5. Средняя производительность разных подходов реализации алгоритма BFS

На графике (см. Рис.5) изображены кривые средней производительности для следующих алгоритмов:

- dynamic parallelism – алгоиртм GPU со всеми оптимизациями и использованием динамического параллелизма архитектуры Кеплер (1)
- cache && restrict on — алгоритм GPU со всеми оптимизациями (2);
- cache off — алгоритм GPU без оптимизации перестановок для улучшенного кэширования (3);
- restrict off — алгоритм GPU без оптимизации текстурного кэша (4);
- cache && restrict off — базовый алгоритм GPU без оптимизаций (5);
- CPU — базовый алгоритм на центральном процессоре.

Так как в данном алгоритме не нужно хранить веса ребер, то из-за того, что массив вершин для графов размерности до $2^{20}$ целиком помещается в кэш, оптимизации в виде перестановки данных и использования текстурного кэша не дают существенного улучшения. Если массив вершин не помещается в кэш GPU, то использование перестановки элементов и текстурного кэша дает 3-4 кратный прирост производительности.

Наибольший эффект достигается при выполнении начальных и конечных итераций разметки графа с использованием динамического параллелизма, когда количество обрабатываемых данных не велико, и выполнении оставшихся итераций с описанными выше оптимизациями, когда количество обрабатываемых данных на столько много, что применение динамического параллелизма не будет оправданно.

В итоге, удалось достичь средней производительности в 5,5—5,8 GTEPS. Почти пиковая средняя производительность достигается на графах с количеством вершин $2^{20}$ —$2^{23}$, что довольно не плохо для данной архитектуры. Максимальная производительность получена на графе с количеством вершин $2^{21}$ и равна 5,8 GTEPS.

**Заключение**

В результате проделанной работы были разработаны и реализованы алгоритмы для решения на GPU следующих задач: Single Shortest Source Path problem (SSSP) – задачи поиска кратчайших путей от заданной вершины до всех остальных во взвешенном графе и Breadth First Search (BFS) – задачи поиска в ширину в неориентированном графе. В реализации данных алгоритмов были использованы архитектурные особенности Kepler, недоступные в GPU поколения Fermi. Поэтому не удалось провести сравнение двух архитектур — Kepler и Fermi.

Реализованные алгоритмы показали хорошую эффективность на одном GPU. В дальнейшем планируется исследование и реализация данных алгоритмов на многих GPU, так как для обработки графов с количеством вершин $2^{25}$ и более требуется больше памяти.

ЛИТЕРАТУРА:

1. Алгоритм Дейкстры // URL:http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part2.pdf (дата посещения 31.05.2014)
2. Алгоритм Форда–Беллмана // URL:http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part2.pdf (дата посещения 31.05.2014)
3. Harish, P. and Narayanan, P.J. «Accelerating large graph algorithms on the GPU using CUDA» // Proceedings of the 14th international conference on High performance computing (Berlin, Heidelberg, 2007), 197–208.
4. Черноскутов М.А. «Сравнение скорости работы GPU и CPU при решении задач класса «data intensive». // Труды Международной суперкомпьютерной конференции «Научный сервис в сети Интернет: экзафлопсное будущее» сентябрь 2011, с. 731-735
5. Takaaki Hiragushi, Daisuke Takahashi «Efficient Hybrid Breadth-First Search on GPUs.» // ICA3PP 2013, 40-50.
6. Pedro J. Martín, Roberto Torres, Antonio Gavilanes «CUDA Solutions for the SSSP Problem.» // ICCS 2009, 904-913
7. Архитектура Nvidia Kepler // URL:http://www.nvidia.com/content/PDF/kepler/NVIDIAKepler-GK110-Architecture-Whitepaper.pdf/ (дата посещения 31.05.2014)
8. Знакомство с NVIDIA CUDA // URL:www.render.ru/books/show_book.php?book_id=840 (дата посещения 31.05.2014)
9. Боресков А.В. , Харламов А.А. Основы работы с технологией CUDA, Изд. ДМК Пресс, Москва, 2010г, 234стр.
10. D Chakrabarti, Y Zhan, and C Faloutsos. «R-MAT: A Recursive Model for Graph Mining.» // In Proceedings of 4th International Conference on Data Mining, pages 442-446, 2004.
11. Murphy R., Wheeler K., Barrett B., Ang J. «Introducing the Graph 500» // Cray User's Group (CUG), May 5, 2010.
12. NVIDIA GTX Titan // URL:http://www.nvidia.ru/object/geforce-gtx-titan-ru (дата посещения 31.05.2014)
13. Graph 500 Benchmark 1 (“Search”) // URL:http://www.graph500.org/specifications (дата посещения 31.05.2014)
14. Hong, S. et al. «Accelerating CUDA graph algorithms at maximum warp» // Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (New York, NY, USA, 2011), 267–276.
15. Merrill D., Garland M., Grimshaw A. «Scalable GPU graph traversal»// Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP’12), 117-128.